编译原理实验二　实验报告

组长：白家杨 161220002 组员：段高磊 161220036

一、编译说明

本次实验不改变助教的Makefile文件，编译方法为直接在Code文件夹下执行make指令。想要检测特定的程序文件，比如example.cmm，步骤为：先执行make编译（生成名为 **parser** 的可执行文件），然后执行以下命令（假定example.cmm文件与parser在同一目录下）：

```
./parse example.cmm
```

二、功能实现

1. 必做功能：基于C--语法的7个假设，对输入的文件进行语义分析，检查指定的17种错误类型

2. 选作**任务2.2**：修改C--语言假设4，针对变量增加**嵌套作用域**；在新的假设下，完成17种错误类型检查

三、实现过程

1. 类型的表示

我们选用书中定义的**Type**结构与**FieldList**结构来表示C--语言中的类型；并额外定义了**VarObject**结构与**FuncObject**结构来分别描述变量与函数的性质——**VarObject**结构中包括名称，类型以及指示是否为左值的布尔变量；**FuncObject**结构中包括函数名称，函数返回类型以及 记录函数形参列表的链表（详情见Code/symboltable.h）。

2. 符号表的确定

我们选择**散列表**作为符号表的具体存储结构，并采用开散列方法（或者拉链法）来处理冲突；至于散列函数，我们选取**P.J.Weinberger** 提出的hash函数（详情见Code/symboltable.c中的**pjwhash函数**）。我们一共创建维护了两张散列表，结构分别如下：

```
typedef struct ValHashTable
{   int depth; //刻画变量所在作用域的深度
    VarObject* val; //描述变量或者结构体
    struct ValHashTable *indexNext; // same index after hashing
    struct ValHashTable *fieldNext; // same field in the syntax tree
} ValHashTable;
```

```
typedef struct FuncHashTable
{   FuncObject* func; //描述函数
    struct FuncHashTable *indexNext; // same index after hashing
} FuncHashTable;
```

第一张表储存所有的变量以及结构体，**val**为VarObject结构体定义的变量；第二张表储存所有的的函数，**func**为FuncObject结构体定义的变量。

3. 多层作用域的处理（任务2.2）

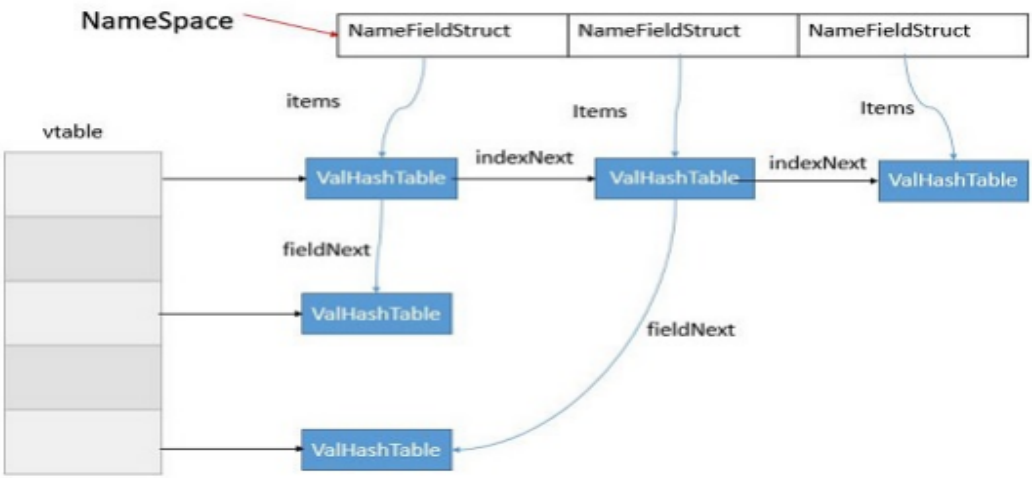
我们使用实验教材所说的Imperative Style来维护支持多层作用域的符号表。在前面1中已经分别建立了变量符号表和函数名符号表，我们这里用一个类似于栈的结构指向不同的作用域的变量。栈的结构如下所示：

```
typedef struct NameFieldStruct
{   int deep;//表示该作用域属于的层数
    int size;//当前作用域下变量的数目
    ValHashTable*items;//指向当前作用域下最新加入的变量
    NameFieldStruct* next;//指向上一层的作用域。
} namesfield;
//这里负责有关命名空间的处理,作用域
NameFieldStruct* NameSpace;
unsigned int CurrentDept;
```

上面*NameSpace*表示指向栈顶的指针，*CurrentDept*表示当前作用域层数。每次访问*NameSpace*即可获得当前作用域的信息。

命名域的初始化和符号表的初始化同时进行。在命名域的初始化封装在**initNameSpace()**中，函数中将*CurrentDept*初始化为0，新建一个新的结构体**NameFieldStruct**，里面的*deep*、*size*均为0，*items*、*next*为空，NameSpace指向这个结构体，表示全局作用域下，层数为0，当前变量数为0，尚未加入变量，指向更上一层的指针为空。

在变量需要加入符号表的时候，首先建立一个**ValHashTable**指针指向新建的一个ValHashTable结构，这个结构的val就指向需要加入的变量对象。然后在 **AddToValHashTable(ValHashTable* table_item)** 中将这个**ValHashTable**指针作为传入参数，实现将该**ValHashTable**对象加入vtable中，这时**ValHashTable**的*indexNext*指向散列表vtable中同散列函数值的下一个变量。最后执行 **AddToNameSpace(table_item)** 将这个**ValHashTable**放入当前作用域 *NameSpace*，即将**ValHashTable**的*fieldNext*指向*NameSpace*的*items*，并将*NameSpace*的*items*指向当前**ValHashTable**，最后更新当前作用域的*size*。最终实现出来的数据结构应当如下图所示：



4. 错误检查说明

本次实验在实验一构建的语法树上完成，所以我们需要对syntax.y中的每一个产生式都书写一个处理函数；为了方便起见，我们取**每个非终结符的名称作为其处理函数的名称**，详见Code/semantic.h中的函数声明。

- 类型1~2：在函数**Exp**中进行判断——提取使用的变量或函数名，检查在对应的散列表中是否有相应的定义。
- 类型3~4：在函数**VarDec**中进行判断——提取要定义的变量名，检查ValHashTable中是否已经有同名的变量或结构体定义（在嵌套作用域下，检查ValHashTable中同一作用域中变量，即depth相同）；提取要定义的函数名，检查FuncHashTable中是否已经有同名的函数定义。
- 类型5：在函数**Dec**（初始化）以及函数**Exp**中进行判断：提取等号两边对象的类型进行判断。
- 类型6：在函数**Exp**中进行判断——判断等号左边对象的lvalue是否为真。
- 类型7：在函数**Stmt**中进行判断：提取IF语句以及WHILE语句中的条件，判断是否为int；在函数**Exp**中进行判断：判断逻辑操作两边是否均为int，判断关系操作以及算术操作两边是否同为int或float。
- 类型8：在函数**Stmt**中进行判断：提取词法单元RETURN后对象的类型进行判断。
- 类型9：在函数**Exp**中进行判断：在FuncHashTable中获取相应函数的形参列表，忽略形参名进行判断（先判断数目是否一致，若一致再依次判断类型）。
- 类型10，11，13 <非法使用“[...]”、“(...)”及“.”>：在函数**Exp**中进行判断：判断词法单元LB左侧的对象是否为数组，判断词法单元LC左侧的对象否为函数，判断词法单元DOT左侧的对象否为结构体。
- 类型12：在**VarDec**（定义）以及**Exp**（使用）中判断：提取“[...]”中的对象，判断类型是否为整数。
- 类型14~17 <与结构体有关的错误的判断>：分别在使用结构体变量，定义结构体时进行判断。

额外说明：

- 对于**错误类型7**中IF语句与WHILE语句中的条件只为int型的解释：只要最终的结果为int型即可，如

  ```
  1 if(1.5 > 1.0) ...;
  ```

  此时不报错，因为我们允许关系运算符两边同为int型或float型——运算结果为true，返回1；结果为false，返回0；返回的对象为int型。

- 我们允许同行报多个错误，如

  ```
  1 int func(){
  2     float x = y + 1.0;
  3 ...}
  ```

  此时在第二行可以报：

  ```
  1 Error type 1 at Line 2: Undefined variable "y".
  2 Error type 7 at Line 2: Type mismatched for operands.
  3 Error type 5 at Line 2: Type mismatched for assignment.
  ```

  因为y未定义，所以类型不可知（本质错误），由此可能引发的操作符不匹配以及赋值号两边类型不匹配错误，我们认为这是合理的，所以也将结果呈现出来；此外，我们也能够保证同行中一定输出本质错误。